Performance of Service-Discovery Architectures in Response to Node Failures

C. Dabrowski, K. Mills, and A. Rukhin

U.S. National Institute of Standards and Technology
Gaithersburg, MD 20899

cdabrowksi@nist.gov

ABSTRACT

Designs for distributed systems must consider the possibility that failures will arise, and must adopt specific failure detection and recovery strategies. Many distributed-object systems employ simple techniques to detect and report failures, requiring applications to decide upon appropriate recovery strategies. In this paper, we investigate the ability of selected designs for service-discovery protocols to detect and recover from failure of remote services when used to support real-time distributed control applications. We model two architectures (two-party and three-party) underlying most commercial service-discovery systems. We use simulation to quantify functional effectiveness and efficiency achieved by the two architectures as the rate of failure increases for remote services. We further decompose non-functional periods into failure-detection latency and restoration latency. Our quantitative measurements suggest that a two-party architecture yields better robustness than a three-party architecture. We discuss the underlying causes for this outcome.

1. INTRODUCTION

Designs for distributed systems must consider the possibility that failures will arise, and must adopt specific failure detection and recovery strategies [1]. Much existing research surrounding failures in distributed systems focuses on providing fault-tolerant invocation of remote methods, either through parallel execution of replicated components or through automated checkpoint and restart procedures [2-4]. Fault-tolerant remote-method invocation typically relies upon a layer of mechanisms to detect and recover from failures without requiring application-specific awareness or action. While such application-transparent fault-tolerance appears appealing, many current distributed object systems, even large systems, employ simpler techniques that detect and report failures, requiring applications to decide upon appropriate recovery strategies [5-7]. In this paper, we investigate one such set of simpler techniques requiring application awareness and cooperation. These techniques encompass the fundamental failure detection and recovery strategies available in service-discovery systems [8-13].

In previous work, we investigated change propagation in various service-discovery systems under communication failure [14] and message loss [15]. Our investigations yielded quantitative measures for the effectiveness, responsiveness, and efficiency of alternate system designs. In this paper, we investigate the effectiveness, efficiency and latency of service-discovery systems in detecting component failure and locating replacements. We model specific discovery strategies and failure-recovery techniques in combination with two major architectural variants found in service-discovery systems: two-party, where clients and services rendezvous directly, and three-party, where clients and services rendezvous through a directory. For the three-party architecture, we consider topologies that include directory replicas. Our models, which adapt discovery and recovery strategies from the Jini™ Networking Technology [10] and Universal Plug-and-Play [9] specifications, layer a real-time distributed control application above each of the discovery systems. We model application-level strategies that focus our experiments on the fundamental properties of service-discovery protocols; thus, we exclude a number of possible application choices, such as service caching. We measure functional effectiveness, defined as the proportion of time that a distributed application meets its requirements, or more precisely, as the proportion of time that a client component possesses an operational set of remote services needed to accomplish its task. To provide a clear picture of failure response, we also measure both failure-detection latency (time required to recognize that a remote service used by the client has failed) and failure-recovery latency (time required for the client to replace a failed service). We also measure overhead as the number of messages sent. Our models are written using Rapide [16], which records complete event traces that permit detailed analysis of system behavior, helping us to determine causes underlying quantitative performance.

2. DISCOVERY AND RECOVERY

Service-discovery protocols enable networked components to rendezvous and to combine with discovered components into distributed applications meeting specific requirements. Discovery protocols include f*ailure-detection and recovery techniques* that enable components within distributed applications to detect and react to failures by restoring communications with remote components or by locating alternate components. A number of different designs have been proposed for service-discovery systems. For example, a team at Sun Microsystems designed Jini Networking Technology, a general service-discovery system atop Java^{TM}. As another example, a group from Microsoft and Intel conceived Universal Plug-and-Play (UPnP) to provide plug-and-play components for distributed systems.

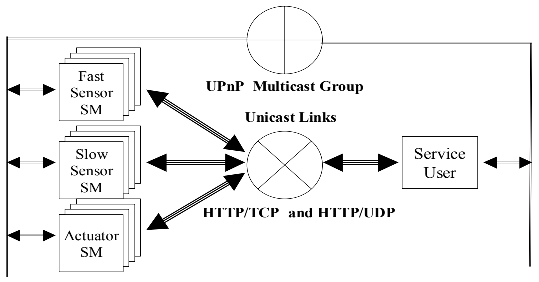

Figure 1. Two-party service-discovery architecture with one service user and 12 service managers

2.1 Service Discovery

Our analysis of six discovery systems [8-13] revealed that most designs use one of two underlying architectures: two-party or three-party. A two-party architecture consists of two component types: service manager (SM) and service user (SU). The three-party architecture adds a third component type, service cache manager (SCM). Multiple SCMs can be used to mitigate the effect of SCM failure. In both architectures, service discovery occurs passively, via multicast announcements, and actively, via multicast queries. Each SM maintains a database of service descriptions (SDs), where each SD encodes the essential characteristics of a particular service provider (SP) managed by the SM. Each SU seeks SDs satisfying specific requirements. Where employed, the SCM operates as an intermediary, matching advertised SDs of SMs to SD requirements provided by SUs.

In this study, each SM manages one SP from among three service types: fast sensor, slow sensor, and actuator. Our experiment consists of four instances of each service type, whose roles are explained below. Figure 1 shows a two-party architecture deployed in our experiment topology with 12 SMs and one SU. To animate our two-party model, we incorporated discovery behaviors from the UPnP specification, as described elsewhere [14, 15]. Figure 2 shows the three-party architecture in our experimental topology: with 12 SMs, one SU, and up to three SCMs. To animate our three-party model, we chose discovery behaviors from the Jini specification, as described elsewhere [14, 15].

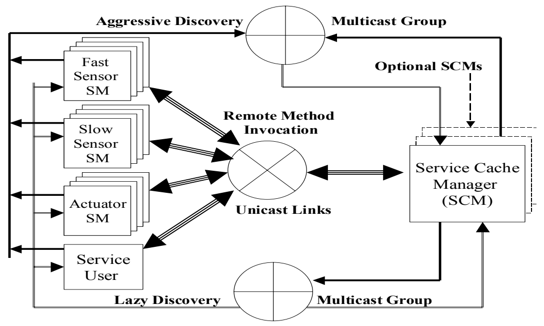

Figure 2. Three-party service-discovery architecture with one service user, 12 service managers, and up to 3 service cache managers

2.2 Failure-Detection Techniques

To detect failures, applications using discovery systems rely on a combination of two techniques: monitoring periodic transmissions and retrying ad hoc transmissions (where exceeding a retry bound causes an exception). Discovery protocols specify periodic transmission of key messages. In addition, components employing remote services may maintain regular contact to accomplish application-specific tasks. Components can listen for these recurring messages, much as a heartbeat can be monitored to assess patient health. For example, both Jini and UPnP periodically announce resource availability. Similarly, a sensor service may periodically issue readings to its clients. Failure to receive scheduled communications might indicate that the remote service has failed, or that the channel between client and service is blocked. In other situations, software components send messages using reliable communication protocols, which persistently resend unacknowledged messages up to some bound, issuing a remote exception (REX) if the bound is exceeded. For example, a client may attempt to invoke a method offered by a remote service that has failed. In the three-party architecture, a SU might attempt to query for a SD from a failed SCM, only to receive a REX. Failure detection enables components to employ recovery techniques.

2.3 Failure-Recovery Techniques

Discovery systems generally support two recovery techniques: soft-state and application-level persistence. Periodic announcements issued by a component convey *soft* information about component state, which a receiver can cache for a period of time, consistent with the expected announcement rate. Each new announcement may convey updated state information; thus, a receiver overwrites previously cached state with state from newly arriving announcements. When an announcement fails to arrive, a receiver discards previously cached state, effectively eliminating knowledge about existence of the announcing component. When announcements resume, a receiver rediscovers the remote component and recovers the latest component state. Our application uses a modified form of soft state, which allows discarded components to be either rediscovered or replaced. For example, upon failure of heartbeat messages sent by UPnP SMs to refresh cached SDs, a SU discards knowledge of the SM and any associated SDs. Similarly, a SU may discard knowledge of a SM and SD for a remote sensor upon failure to receive sensor updates. To effect recovery, UPnP SUs may commence periodic multicast (*Msearch*) queries to search for a new instance of a required service. Once the SU regains a SD meeting requirements, the related queries cease. In Jini, loss of contact with a service may cause the SU to query a SCM for a replacement. In addition, service unavailability may be indicated by failure of heartbeat messages sent by Jini SMs to refresh SDs cached on SCMs, causing the SCM to discard the SD and to notify SUs that indicated interest in learning about service failures. Periodic announcements ensure rediscovery of the SCM by SMs within 120s after the SM recovers. The Jini SU can then receive the corresponding SD through notification or query. Of course, in Jini, SCMs could also fail. SCM startup announcements ensure rediscovery of a restarted SCM within 30s.

When failures lead to a REX, discovery systems generally expect application software to initiate recovery, guided by an application-level persistence policy, which might ignore the REX, retry the operation for some period, or discard knowledge of the remote component. Since our experiment simulates a real-time control application, we chose not to persist after a REX, but instead to discard knowledge of the associated remote component, relying on periodic announcements and soft state to recover. This policy is also used in the three-party model when SCM failure is detected through a REX in response to a query (SU) or registration refresh (SU or SM). After discarding knowledge of a SM (UPnP) or SCM (Jini), all operations involving the remote component cease.

3. EXPERIMENT DESCRIPTION

We investigate how effectively the two alternate service-discovery architectures, and associated failure detection and recovery mechanisms, provide clients with required services as nodes hosting the services fail and recover. We model the two- and three-party architectures using the four topologies shown in Figures 1 and 2. In all topologies, we deploy a single SU and twelve SMs, where each SM manages a specific type of SP: "fast" sensor, "slow" sensor, or actuator. The twelve SMs include four of each SP type. After discovery and activation by the SU, a "fast" sensor transmits a reading every two seconds and a "slow" sensor transmits a reading every 30 seconds. Once discovered and activated by the SU, an actuator can be invoked after the SU receives an appropriate combination of readings from a "fast" and "slow" sensor. In our experiment, we simulate actuation attempts using a uniform distribution with a mean of 60s. When the SU holds one SD for a SP of each type ("fast" sensor, "slow" sensor, and actuator) and each of the SPs is operational, then the application is considered *functional*. If the SU lacks SDs for one or more SP type or if one or more of the SDs held by the SU describes a SP that is not operational, then the application is considered *non-functional*. The experiment measures accumulated *functional time* in proportion to a duration *D* during which SMs and SCMs periodically fail and recover. To establish initial conditions, each topology is exercised until discovery completes, and the application becomes functional. To focus exclusively on failure detection and recovery processes, we do not cache services; the SU holds at most one SD for each SP type at any time. In the three-party architecture, some additional decisions are necessary. For each SD discovered and retained, the SU registers with the SCM for notification about failures. The SU refreshes notification registrations every 300s. Each SM registers with each discovered SCM, and refreshes every 60s (slow sensors/actuators) or 300s (fast sensors).

3.1 Failure Model

During *D*, each SM (and SCM in the three-party case) fails randomly and independently, although at least one service of each type always remains active so that the application could become functional. We calculate a mean time to failure, *MTF*, from a failure rate *R*, varied from 0.1 to 0.9 of *D* in 0.1 increments, where $MTF = (1 - R) * D$. We choose node failure times from a "stepped" normal distribution with three steps: a 0.15 probability that failure occurs before ($MTF - 0.2 * MTF$), a 0.7 probability that failure occurs between ($MTF - 0.2 * MTF$) and ($MTF + 0.2 * MTF$), and a 0.15 probability that failure occurs between ($MTF + 0.2 * MTF$) and ($2 * MTF$). Failure time is distributed uniformly within each step.

When a SM or SCM fails, affected services become unavailable for a time. There are three failure classes, each with a different probability, *P*, and duration. Short failures occur with $P = 0.1$ for a fixed duration (135s); intermediate failures occur with $P = 0.7$ for a duration selected uniformly on the interval 180-300s, long failures occur with $P = 0.2$ selected uniformly on the interval 480-600s.

3.2 Metrics

We define non-functional time, *NF*, as accumulated time during which an application is in a non-functional state. Assuming we can measure *NF*, over a given duration *D*, then functional effectiveness, *F*, can be quantified as a ratio: $F = (D - NF)/D$. We define consistency conditions to measure *NF*, as explained below.

A client in a distributed application may become non-functional due to failure of remote components but incur a delay before detecting the failure. We call this delay failure-detection latency. After detecting a non-functional state, the application may incur some delay while restoring required services. We call this delay failure-recovery latency. During periods when a client incurs either failure-detection or failure-recovery latency or both (the states can overlap when a client requires more than one remote service), the distributed application is non-functional. We accumulate such non-functional periods to *NF*.

We define two consistency conditions such that violation of one corresponds to failure-detection latency and violation of the other corresponds to failure-recovery latency. The following consistency condition requires each SD held by a SU to match a SD managed by a SM. More formally,

$$\forall[SM, SU, SD]\ (SM, SD) \in discoveredServices_{SU}$$
$$\rightarrow \exists SM \bullet SD \in managedServices_{SM}$$

This condition (**CC-1**) is violated (and failure-detection latency commences) when a SM fails but the SU holds a SD provided by the SM. Once the SU discards the SD, or the SM recovers, consistency is restored (and failure-detection latency ends). A second consistency condition requires that available SDs matching SU requirements should be known to the SU. More formally,

$$\forall[SM, SU, SD]\ SD \in managedServices_{SM} \wedge$$
$$SD \in resourcesNeeded_{SU}$$
$$\rightarrow (SM, SD) \in discoveredServices_{SU}$$

This condition (**CC-2**) is violated (and failure-recovery latency begins) after the SU purges a SD for a failed service and commences search. Consistency returns (and failure-recovery latency ends) when the SU finds a SD matching its needs.

4. RESULTS AND DISCUSSION

For each of four topologies (two-party and three-party with one, two, and three SCMs), we set $D = 1800$s and executed multiple repetitions for each value of *R* using the failure model described in 3.1. We conducted separate experiment runs for cases where failed nodes (including SMs and SCMs) are discarded and *replaced* by new nodes, and for cases where failed nodes *restart*, maintaining persistent information in the manner specified by the protocols. For the replacement case, we ran a second variant of the experiment where all SMs for a resource type may fail. We recorded functional effectiveness, detection latency, recovery latency, and the total number of messages exchanged in each run.

4.1 Effectiveness and Efficiency

Figure 3 shows average functional effectiveness of the two-party and three-party architectures for the replacement case as *R* increases, and where one SM for each service type is always available (implying that the system could be functional for all of *D*). In examining Fig. 3, recall how failure detection occurs. In the two-party model, the SU may detect service unavailability by monitoring cyclical sensor readings or by monitoring notification registration refreshes. In the three-party model, the SCM notifies the SU if the SM fails to refresh service registrations. In both models, the SU may also detect unavailability when a REX occurs in response to attempted actuations. To become functional again, the SU must invoke appropriate recovery mechanisms to regain SDs to replace unavailable services. In the three-party architecture, at least one SCM must be operational for recovery to succeed. During periods when all SCMs fail, the SU is unable to recover needed services, increasing non-functional time.

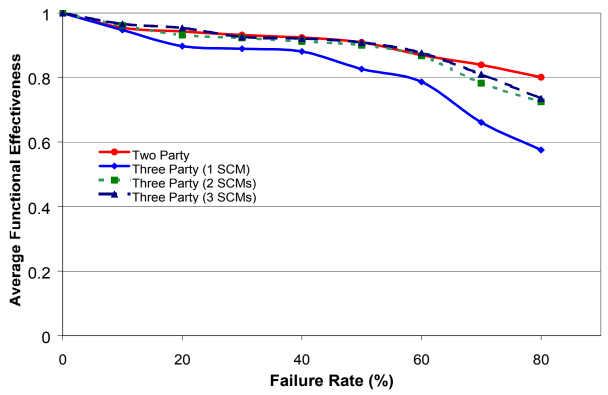

Figure 3. Functional effectiveness for four topologies under increasing *R* for the replacement case where at least one SM of each type is operational (60 reps/point)

Overall, the two-party architecture proves more effective above 60% failure rate, allowing the SU to remain functional for as much as 80% of *D* even when the failure rate reaches 80% (*MTF* = 360s). At rates below 60% the effectiveness of two-party is comparable to three-party with two and three SCMs. Fig. 3 also shows that effectiveness improves for the three-party architecture as the number of SCMs increase, though even with 3 SCMs, performance does not equal that of the two-party architecture. Adding SCMs improves effectiveness by lowering the incidence of concurrent failure of all SCMs.

Message counts (Fig. 4) reveal the two-party architecture to be significantly more efficient than the three-party architecture. Note also that for the three-party architecture, total message counts decrease as failure rate increases, because SCMs remain down for longer periods; thus, requiring fewer registration refresh and SCM heartbeat messages. For the two-party model, message counts increase slightly at high failure rates because the SU invokes active recovery procedures after detecting failures. Fundamentally, the three-party architecture relies on redundancy of SCMs to improve functional effectiveness; thus, exacting a high overhead at low failure rates, but permitting overhead to diminish as failure rate increases. The two-party architecture relies on active recovery invoked by a SU; thus, at low failure rates overhead is lower because recovery procedures are not invoked often, but overhead increases with failure rate as recovery procedures are invoked more often.

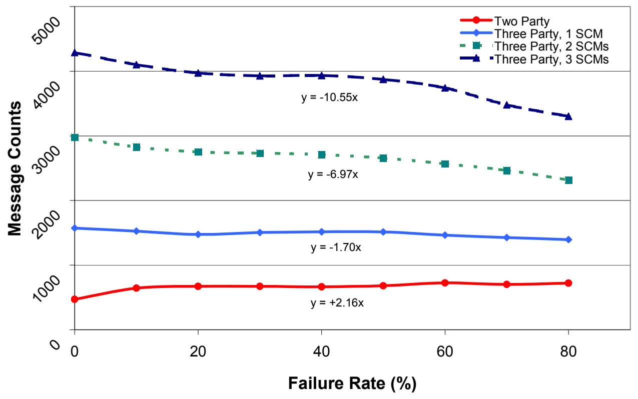

Figure 4. Average message counts for four topologies under increasing *R* for the replacement case where at least one SM of each type is operational (30 reps/point)

4.2 Underlying Causes

To better understand differences in effectiveness among the alternate architectures, we decomposed non-functional time to show the estimated proportion attributable to failure-detection latency and to failure-recovery latency. Figure 5 shows that detection latency is the dominant (~80%) component of non-functional time for the two-party model. Analysis of execution traces using the Rapide toolset showed most failures were detected through missed sensor readings (2s for fast sensors and 30s for slow sensors) or REXs received in response to failed actuations. We suspected that in the two-party architecture detection latency, and therefore non-functional time, could be reduced by increasing registration-refresh frequency; thus, decreasing the interval between heartbeats. Failed notification refresh attempts by the SU would permit detection of SM unavailability (and violation of **CC-1**) before non-receipt of slow sensor readings or failed actuation attempts. To test this theory, we lowered the registration refresh frequency from 300s to 30s in the two-party model, and reran the experiment The result was a 49% drop in detection latency leading to a 2.6% overall improvement in functional effectiveness (an increase in the mean effectiveness across all failure rates from 0.908 to 0.932). However, efficiency decreased 69%, with a rise in message count from an average of 662 to 1116. Similarly in the three-party architecture, we suspect increasing refresh frequency for service registrations would lead to earlier detection by the SCM of SM failure [see 17], and to earlier notification for the SU. Of course, increasing the heartbeat rate also would decrease efficiency.

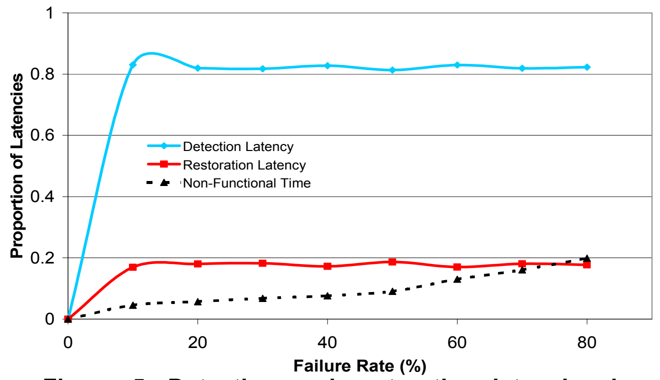

Figure 5. Detection and restoration latencies in two-party service-discovery model as a proportion of non-functional time (also shown) (60 reps/point)

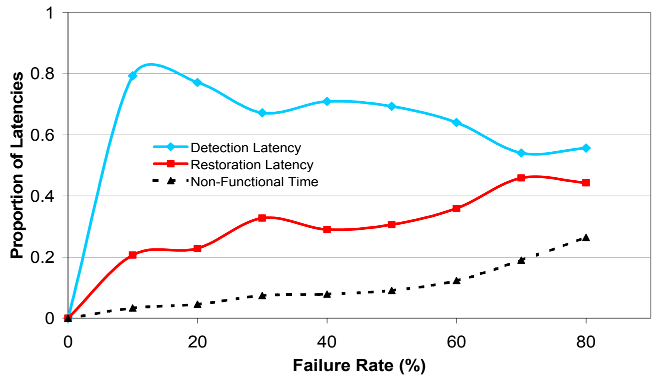

Figure 6. Detection and restoration latencies in three-party service-discovery model with 3 SCMs as a proportion of non-functional time (also shown) (60 reps/point)

Our data for the three-party architecture show that above 60% failure rate the incidence of concurrent failure of all SCMs increases steadily. This precludes finding available services meeting SU requirements; thus, leaving the system in violation of **CC-2**. To restore consistency and achieve operational functionality, a SCM must first recover, accept registrations for the SU and available SMs, and then propagate matching SDs to the SU. Lacking an ability to directly discover SMs, the SU remains non-functional while awaiting recovery of at least one SCM. These effects are evident in Fig. 6, which shows the proportion of recovery latency increasing for the three-party model (3 SCMs) as the failure rate rises. This trend is more marked as the number of SCMs decreases (not shown here). We speculate that functional effectiveness might improve for the three-party model if SUs were permitted to discover SMs directly when no SCMs are available. We plan experiments along these lines using the Service Location Protocol (SLP) [12], which enables switching between the two- and three-party architecture as the situation warrants.

4.3 Results for Experiment Variants

To confirm our findings, we varied the experiment in two respects. First, we changed node behavior to allow failed nodes to restart rather than be replaced by new nodes. In this case, three-party SCMs that recovered were allowed to retain previous, unexpired service registrations and notification registrations in accordance with the Jini protocol, while two-party SMs were permitted to retain notification registrations. The results showed no significant differences in performance between the restart and replacement cases, the graphs (not shown) were almost identical. This occurs in the three-party case because most of the persistent registrations expire by the time a failed SCM restarts. In the two-party case, where only notification registrations persist, the SU that registered the notification is likely to have discarded knowledge of the SM by the time it restarts. Since, in our experiment, restarting nodes derive little value from persistent information, functional effectiveness is mainly influenced by soft-state mechanisms, as in the replacement case.

Second, we varied the experiment to permit all SMs to fail, rather than to have at least one SM always available for each service type. The graphs (not shown) illustrate functional effectiveness for both the two- and three-party models decreases substantially above $R = 60\%$, as the incidence of concurrent SM failures increases, resulting in extended periods when no SMs were available for a service type needed by the SU. Though the absolute functional effectiveness declined, the ranking of the curves remained the same as in the previous experiments, with the two-party model proving most effective followed by the three-party model with three-, two-, and one-SCM topologies, respectively. Thus, in all of our experiment variants, the two-party model achieved better functional effectiveness than the three-party model.

5. CONCLUSIONS

This study provides an initial characterization of the performance of service-discovery architectures in response to node failures, which complements our previous studies of response to communication failures and message loss. The present study shows that in response to node failure, two-party systems exhibit better functional effectiveness and efficiency than three-party systems, with three-party SCMs being a potential point of vulnerability. Possible solutions to mitigate this vulnerability require further study. Similarly, further research is needed to verify that registration refresh rates or service caching could improve functional effectiveness. Finally, we need to verify that our conclusions hold in networks with large numbers of services.

6. ACKNOWLEDGMENTS

The work discussed in this paper was funded in part by DARPA, under the auspices of the FTN and DASADA programs.

7. REFERENCES

[1] G. Bieber and J. Carpenter, "Openwings A Service-Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems," on the http://www.openwings.org web site.

[2] Fault Tolerant CORBA Specification, v1.0, ptc/00-04-04, Object Management Group.

[3] C. Marchetti, A. Virgillito, and R. Baldoni, "Design of an Interoperable FT-CORBA Compliant Infrastructure," *Proceedings of the European Research Seminar on Advances in Distributed Systems* (ERSADS), 2001.

[4] D. Liang et al., "A Fault-Tolerant Object Service on CORBA," *The Journal of Systems and Software*, vol. 48, 1996.

[5] Y.M. Wang, O.P. Damani, and W.J. Lee, "Reliability and Availability Issues in Distributed Component Object Model (DCOM)," *Proceeding of the International Workshop on Community Networking*, 1997, pp. 59-63.

[6] Felber, P. et al. Failure Detectors as First Class Objects, *Proceedings of the International Symposium on Distributed Objects and Applications* (DOA'99), IEEE Computer Society Press, September 5-7, 1999, p. 132.

[7] Carey, R.W. et al. "LARGE-SCALE CORBA-DISTRIBUTED SOFTWARE FRAMEWORK FOR NIF CONTROLS", *Proceedings of the 8th International Conference on Accelerator & Large Experimental Physics Control Systems*, Stanford Linear Accelerator Center, November 27-30, 2001, p. 425.

[8] Salutation Architecture Specification, Version 2.0c, Salutation Consortium, June 1, 1999.

[9] Universal Plug and Play Device Architecture, Version 1.0, Microsoft, June 8, 2000.

[10] Ken Arnold et al, The Jini Specification, V1.0 Addison-Wesley 1999. Latest version is 1.1 available from Sun.

[11] Specification of the Home Audio/Video Interoperability (HAVi) Archiecture, V1.1, HAVi, Inc., May 15, 2001.

[12] Guttman, E., Perkins, C., Veizades, J., and Day, M. Service Location Protocol Version 2, Internet Engineering Task Force (IETF), RFC 2608, June 1999.

[13] Specification of the Bluetooth System, Core, Volume 1, Version 1.1, the Bluetooth SIG, Inc., February 22, 2001.

[14] Dabrowski, C. Mills, K., and Elder, J. "Understanding Consistency Maintenance in Service Discovery Architectures During Communications Failure", *Proceedings of the 3rd International Workshop on Software Performance*, ACM, July 2002, pp. 168-178.

[15] Dabrowski, C., Mills, K., and Elder, J. "Understanding Consistency Maintenance in Service Discovery Architectures In Response to Message Loss", *Proceedings of the 4th International Workshop on Active Middleware Services*, IEEE Computer Society, July 2002, pp. 51-60.

[16] Luckham, D. "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events," http://anna.stanford.edu/rapide, August 1996.

[17] Bowers, K., Mills, K., and Rose, S. "Self-adaptive Leasing for Jini", accepted for presentation at *IEEE PerCom 2003*, Fort Worth, Texas, March 2003.